podft: On Accelerating Dynamic Taint Analysis with Precise Path Optimization

Zhiyou Tian
Xidian University
21151213645@stu.xidian.edu.cn

Cong Sun
Xidian University
suncong@xidian.edu.cn

Dongrui Zeng
Palo Alto Networks
dzeng@paloaltonetworks.com

Gang Tan
Pennsylvania State University
gtan@psu.edu

BAR 2023
• Dynamic taint analysis (DTA)
  • What is it?
  • Useful for security

• Binary-level dynamic data-flow tracking (DFT)
  • Dynamic binary instrumentation (DBI)
  • Virtual machine manager (VMM)
  • Emulator
• DBI-based DTA
  • Focus on explicit flows
  • Hold the tainting states within tagging memory

• Challenge of DTA —— significant performance penalty

High Cost!
• Existing works
  • Lift \((MICOR 2006)\)
    • static fast path
  • Libdft \((VEE 2012)\)
    • on Pin
    • DBI inline routines
  • TaintRabbit \((ASIA CCS 2020)\)
    • on DynamoRIO
    • dynamic fast path
  • SELECTIVETAINT \((USENIX 2021)\)
    • static binary rewriting
    • bloat the attack surface
    • value-set analysis
    • cannot work on library code

• **podft advantages**
  • more efficient
  • not bloat the attack surface
  • consider library code
  • flexible scalability

Our work —— podft defines and enforces various fast paths
Design of podft

- podft overview
  - BPA-based CFG Construction
  - VSA-based tainted inst identification
  - Tracking policy construction
  - PDG-based function abstract (from SDFT)
  - Pin-based Tracker

Fig. 1. Framework of podft (dashed block = usage of existing tools)

Next — give a toy example to demonstrate
- **BPA-based CFG Construction**
- **VSA-based tainted inst identification**

**Fig.2 toy example**

```c
void toy_test(int fd, char *buf, int size){
    int read_len = read(fd, buf, size); // taint source
    if(read_len > 0){
        printf("read data: \%s\n", buf);
        for (int i = 0; i<2; i++){
            buf[i] = i;
            write(fd, buf[i], 1); // taint sink
        }
    }
}
int main(int argc, char *argv[]){
    char buffer[64] = {0};
    int fd = open(argv[1], O_RDWR);
    toy_test(fd, buffer, 64);
    return 0;
}
```
**Tracking policy construction**

- naive fast path → `main(bbl-c)`
- complex fast path → `toy_test+0x44(bbl-b)`
- slow path → `toy_test(bbl-a)`
- function fast path → `printf` etc..

---

### Example demonstration

- **Why?**

```
0x80484f4: push ebp
0x80484f8: mov ebp, esp
0x8048500: sub esp, 0x28
0x8048503: mov eax, dword ptr [ebp - 0x10]
0x8048506: mov dword ptr [esp + 8], eax
0x8048509: mov eax, dword ptr [ebp + 0x10c]
0x804850d: mov dword ptr [esp + 4], eax
0x8048511: mov eax, dword ptr [ebp + 0x10]
0x8048514: mov dword ptr [esp], eax
0x8048517: call 0x8048390 <read@plt>
```

```
0x80484f4: push ebp
0x80484f8: mov ebp, esp
0x8048500: sub esp, 0x28
0x8048503: mov eax, dword ptr [ebp - 0x10]
0x8048506: mov dword ptr [esp + 8], eax
0x8048509: mov eax, dword ptr [ebp + 0x10c]
0x804850d: mov dword ptr [esp + 4], eax
0x8048511: mov eax, dword ptr [ebp + 0x10]
0x8048514: mov dword ptr [esp], eax
0x8048517: call 0x8048390 <read@plt>
```

---

**basic block A (bbl-a)**

**basic block B (bbl-b)**

**basic block C (bbl-c)**
Example demonstration

• **Tracking policy construction**

  • naive fast path $\rightarrow$ main(bbl-c)
    • Not contain potentially tainted instructions

```
0x804857f (0x804857f) main

804857f: push ebp
8048580: mov ebp, esp
8048582: push edi
8048583: push esi
8048584: push ebx,
8048585: and esp, 0xffffffff0
8048588: sub esp, 0x60
804858b: mov eax, dword ptr [ebp + 0xc]
804858e: mov dword ptr [esp + 0xc], eax
8048592: mov eax, dword ptr gs : [0x14]
8048598: mov dword ptr [esp + 0x5c], eax
804859c: xor eax, eax
804859e: lea ebx, [esp + 0x1c]
80485a2: mov eax, 0
80485a7: mov edx, 0x10
80485ac: mov edi, ebx
80485ae:mov ecx, edx
80485b0: rep stosd dword ptr es : [edi], eax
```

basic block C (bbl-c)
Example demonstration

- Tracking policy construction

- **complex fast path** → toy_test+0x44(bbl-b)
  - Contain potentially tainted instructions
  - Hot BBL (be executed multiple times)
  - TaintedMem(bbl) ∩ MergedDep(bbl) = ∅.

because

```c
int read_len = read(fd, buf, size); // taint source
if (read_len > 0) {
    printf("read data: %s\n", buf);
    for (int i = 0; i<2; i++){
        buf[i] = i;
        write(fd, buf[i], 1); // taint sink
    }
}
```

The data delivered to the sink are irrelevant to the tainted data from the source.
• **Tracking policy construction**

• **Slow path** $\rightarrow$ `toy_test(bbl-a)`
  - Contain potentially tainted instructions
  - Not hot or $\text{TaintedMem(bbl)} \cap \text{MergedDep(bbl)} \neq \emptyset$

```assembly
0x80484fd: push ebp
0x80484fe: mov ebp, esp
0x8048500: sub esp, 0x28
0x8048503: mov eax, dword ptr [ebp + 0x10]
0x8048506: mov dword ptr [esp + 8], eax
0x804850a: mov eax, dword ptr [ebp + 0xc]
0x804850d: mov dword ptr [esp + 4], eax
0x8048511: mov eax, dword ptr [ebp + 8]
0x8048514: mov dword ptr [esp], eax
0x8048517: call 0x8048390 <read@plt>
```

**because**

basic block A (bbl-a)
• **PDG-based function abstract**
  - Function fast path → printf etc..

• **Pin-based Tracker**
  **Input:**
  - Function-level policies
  - Naive/Complex fast path policies
  - Slow path policies

  **Output:**
  - DTA results

---

Next —— podft’s efficiency and effectiveness.
• **Experimental Settings**
  - Desktop with a 2.8GHz×4 Intel Core(TM) i7-7700HQ CPU, 8GB RAM, and Linux 3.16.0 kernel (Ubuntu 14.04 32-bit).
  - The DBI framework is Pin v2.14, and libdft.

• **Benchmark Programs**

<table>
<thead>
<tr>
<th>Dataset ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>10 SPEC CPU 2k6 benchmarks</td>
</tr>
<tr>
<td>S2</td>
<td>3 server programs, i.e., Nginx (1.22.0), Apache httpd (2.4.7), and MySQL (5.5.62)</td>
</tr>
<tr>
<td>S3</td>
<td>9 CVE programs, as presented in Table V</td>
</tr>
</tbody>
</table>
Evaluations

- **Efficiency of podft**
  - Compare podft’s efficiency with Taint Rabbit, Dytan, Triton, and Taintgrind.

podft achieves **slowdowns of 1.6x** to **27.9x** with an average slowdown of **10.6x**.

podft is more efficient than the other DTA tools.
Efficiency of podft

- Compare podft’s efficiency with SELECTIVETAINT.

podft achieves slowdowns of 1.6x to 27.9x with an average slowdown of 12.5x, and is generally more efficient than SELECTIVETAINT.
Evaluations

• **Effectiveness of podft’s Dynamic Taint Analysis**

---

Real exploits detection by podft
Develop Pintool over podft to track vulnerability of CVEs

---

<table>
<thead>
<tr>
<th>ID</th>
<th>Program</th>
<th>Type</th>
<th>$T_{podft}$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2021-41253</td>
<td>Zydis v3.2.0</td>
<td>H-OF</td>
<td>1.4</td>
</tr>
<tr>
<td>CVE-2019-8354</td>
<td>SoX v14.4.2</td>
<td>H-OF</td>
<td>2.8</td>
</tr>
<tr>
<td>CVE-2018-19655</td>
<td>dcraw v9.28</td>
<td>S-OF</td>
<td>1.7</td>
</tr>
<tr>
<td>CVE-2018-11575</td>
<td>ngiflib v0.4</td>
<td>S-OF</td>
<td>1.4</td>
</tr>
<tr>
<td>CVE-2018-6612</td>
<td>jhead v3.00</td>
<td>H-OF</td>
<td>1.2</td>
</tr>
<tr>
<td>CVE-2017-1000437</td>
<td>Gravity v0.3.5</td>
<td>RCE</td>
<td>9.8</td>
</tr>
<tr>
<td>CVE-2017-14411</td>
<td>MP3Gain v1.5.2</td>
<td>S-OF</td>
<td>2.2</td>
</tr>
<tr>
<td>CVE-2013-2028</td>
<td>Nginx v1.4.0</td>
<td>S-OF</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Work in progress

more scalable and more flexible to be used in traditional DTA

Fig. 1. the workflow of hot BBL embedding

Fig. 2. the workflow of NeuTaint (SP2020)
THANKS

Thanks for listening